2. Lexical basis of AleC++

The basic role of the interpreter is to analyze a string of ASCII characters, group them in symbols and check if the combination of those symbols agrees with the rules of the language in question. Lexical analyzer does the job of grouping up the symbols. AleC++ has inherited the lexical rules of C++ for the most part; the differences will be documented in detail.

The names of Alecsis input files are arbitrary (the name length is determined by the specific operating system,) but they have to have the extension `.ac`. Extension `.hi` is also allowed for compatibility with version 1.0. File name extensions are the following:

`ac` - Alecsis input file
`hi` – Alecsis 1.0 input file (accepted by newer versions, too)
`h` - Alecsis header file (as in C/C++)
`ar` – Alecsis results (Alecsis output file, Agnu input file)
`ao` – Alecsis object-code file (compiled input file)
`as` – Alecsis assembly language file (created by compiler using option `-S`)
`aa` – Alecsis library
`stat` - Alecsis statistics file (creted when command option `-stat` is used)

The preprocessor processes the file by analyzing the lines beginning with the special character '#', and develops all preprocessor macros in the text. The interpreter in fact analyzes the results supplied by the preprocessor (temporary file,) and not the original text.

2.1. Blank space

Blank space, or blank text is the text which does not produce any effect since the interpreter ignores it. It is used for symbol separation, increased readability of the text and documentation purposes. Since AleC++ is a

superset of C++, it is a free format language (all construct can be extended to an arbitrary number of lines.) Empty character string ' ', horizontal tabulator, and a new line are examples of blank space. These characters are not treated as blank space only if they are in between characters ' ' or " " , e.g. if they are a part of character, that is string constants. Beside the above mentioned characters the interpreter treats comments as blank space, as well.

2.1.1. Comments

Comments allow for the documentation of the text. AleC++ supports three types: basic, line and SPICE comments.

Basic comments are an arbitrary text bounded by `/*` and `*/` (C comments.) They can be arbitrarily long, but cannot be nested.

```
/* this is a one-line comment */

/*****************************************
 *   this is a multiple-line comment       *
 *****************************************/

/*   /* this is an error */ */
```

Line comments are inherited from C++, and last until the end of the current line. The text right of character `//` is considered a comment.

```
// this is a line comment
```

The third type of comments (SPICE) are used in a limited number of cases. Since Alescis supports analogue device models of SPICE simulator, AleC++ has a shortcut for the users using libraries of model cards for mentioned components:

```
spice {
* SPICE syntax is valid in between the characters { and } - this
* is a SPICE comment
.model mn nmos ( level=1 vto=0.7v )
}
```

After reserved word spice inside parentheses { and } only SPICE lexical rules are valid. It means that everything starting with '*' is a comment. The lexical analysis goes until the end of the line, which can be continued if '+' sign is in the first column. Characters '(' and ')' are ignored. In the end the line has to begin from the first column. As much as this rule are cumbersome, they allow direct use of hundreds or even thousands of lines of text using SPICE cards, and can significantly shorten the time needed to model the same elements using Alecsis.

2.1.2. Line connections

Sometimes it may be necessary to define a string longer than the length of one line. If character '\' is placed at the end of the first line, it will be ignored along with the end of the line. This results in the merger of two lines, i.e. the string continues starting with the beginning of the next line. Note that standard C offers this option, as well. Using this method, constant strings can be defined across many lines:

```
"this string spans across \
two lines"
```

2.2. Symbols

Now that we have defined blank space the only thing left are symbols. Symbols in AleC++ are:

- key words
- identifiers
- constants
- operators
- separators

2.2.1. Key words

Key (reserved) words have special meaning and are not to be used outside their definition (except within string constants.) Key words in AleC++ represent a amalgam of key words from C++ and a smaller number of ones created for hardware description. Basic (C) key words are shown in Table 2.1.

Table 2.1: Key words in C.

auto	break	case	char	const	continue	default
do	double	else	enum	extern	float	for
goto	if	int	long	register	return	short
signed	sizeof	static	struct	switch	typedef	union
unsigned	void	volatile	while			

In addition to this ones C++ introduced 11 more key words (Table 2.2) Since AleC++ supports C++ syntax for the most part, these are a part of AleC++ syntax, too:

Table 2.2: Key words in C++

class	delete	friend	inline	new	operator	private
protected	public	this	virtual			

AleC++ has key words used for electronic circuit description. These key words are shown in Table 2.3.

Table 2.3: Key words used in AleC++ only.

action	allocate	asm	after	bjt	capacitor	cccs
ccvs	cgen	charge	clone	conversion	current	ddt
diode	eqn	flow	idt	in	inout	implicit
inductor	jfet	lengthof	library	model	module	mosfet
nlcgen	nlgen	nlvgen	node	now	out	options
plot	process	resistor	root	signal	sweep	temp
timing	transport	vccs	vcvs	vgen	vsin	vpwl
wait						

2.2.2. Identifiers

Identifiers are symbols; names of variables, functions, markers, elements, etc. The names can consist of an arbitrarily long number of characters `a-z`, `A-Z`, and `0-9`, as well as '_'. There are two rules to honour:

Identifier cannot begin with a digit;

Number of characters may vary from implementation to implementation (it depends if the identifier appears in the file system, where names are limited to 8-32 characters.) The current version of Alecsis provides for 255 characters.

Examples of identifiers are:

```
i  counter  i1  i123_a  __fetch2  VeryLongButCorrectIdentifier
```

AleC++ is *case-sensitive*, i.e. capital and small letters differ. The exception to the rule is SPICE environment, since SPICE is not case-sensitive.

2.2.3. Constants

Constants store fixed values of numbers, signs, or strings. AleC++ supports 4 types of constants: integer, real, index, and character.

2.2.3.1. Integer constants

Integer constants can have in decimal, octal, or hexadecimal format. Decimal constants represent a sequence of digits `0-9`, bearing in mind that the first digit cannot be `0`. The length restrictions depend upon the actual implementation, but most UNIX computers are AleC++ integers represented stored using 4 bytes. An example of a decimal constant is:

```
1
12
1279
```

but not:

`037` (octal)

`0x22` (hex)

`-2` (expression)

Decimal constants larger than 2 147 483 648 cause error.

Octal constants consist of a sequence of digits `0-7`, bearing in mind that the first digit cannot be `0`. The error will occur if the octal number is greater than `017777777777`.

Hexadecimal constants need to begin with `0x`, or `0X`. They consist of sequence of digits `0-9` and characters `a-f`, or `A-F`. A hexadecimal constant cannot be larger than `0x7fffffff`.

AleC++ does not support unsigned types. ANSI-C suffix type `u` or `U` (from `unsigned`) are not allowed. Since the number of bytes occupied by the types `short`, `int`, and `long` is identical (4 bytes), suffixes `I` and `L` are not supported either. All integer operations in AleC++ are performed as `signed long`. Reader needs to note that the constant `3u` does not mean `(unsigned)3`, but rather `3.0e-6`, because suffix `u` in AleC++ means *micro*.

2.2.3.2. Real constants

The representation of real constants is the same as in C, or C++. These are a few examples of real constants:

```
1.
1.2
.2
.2e-3
1e12
0.22334
1E12
```

AleC++ introduces a concept of *units*, not unlike similar hardware description languages. To simplify writing of physical constants, one can use suffices that denote thousand times smaller or bigger units. Following examples are valid:

`f` or `F`	- 1e-15
`p` or `P`	- 1e-12
`n` or `N`	- 1e-9
`u` or `U`	- 1e-6
`m` (without `M`)	- 1e-3
`k` or `K`	- 1e3
`M`	- 1e6
`g` or `G`	- 1e9
`t` or `T`	- 1e12

Note: An integer constant becomes a real constant if followed by one of the shown suffixes. It means that `1k` means the same as `1.0k` or `1e3`.

A constant can have a user-defined suffix consisting of alphabets `a-z`, `A-Z`, and/or `_`. The purpose of that suffix is to clearly define physical units of measurement and it is ignored in computing. It follows that the constants:

```
 1k      1kohm      1ohm      1.23pF      33MHz      33cycles
```

are written correctly. An integer constant without the suffix, and with a suffix that is not an unit remains an integer constant.

If you write `1Pa` for a pressure of 1 Pascal, AleC++ will understand it as number of 1.e-12, as P is understood as multiplication with 1.e-12. For that reason, be very careful when writing suffixes for physical constants, or better use only suffixes for multiplying (kilo, milli, micro) and omit the physical unit itself.

It should be noted that SPICE units suffixes are appropriate for text marked by the key word `spice`. More information can be obtained from any of the manuals covering SPICE program.

2.2.3.3. Character constants

A character bounded by the apostrophes is a character constant. If that character cannot be displayed or has a special meaning, the *escape* sequence can be used:

```
'c'     'a'     '+'     '\n'     '\\'     '\007'     '\t'
```

Strings are sequences of characters bounded by " ":

```
"string"
"one more"
"string with the escape character for a new line \n"
```

Rules which apply in ANSI-C, or C++ , apply in AleC++. Note: AleC++ merges all strings separated by a blank space (ANSI-C), e.g.

```
"first and "   "second"
```

merge into

```
"first and second"
```

2.2.3.4. Index (enumeration) constants

As in C, enumeration constants are declared using the key word `enum`:

```
enum Bool { False, True };
```

Constant `False` has the value of `0`, and constant `True` is `1`. The values increase by one starting from `0`, as the new symbols are added. If this setup is not satisfactory in a special case a direct intervention is possible:

```
enum Bool { False, Fatal = 0, True, OK = True };
```

Symbols without the initial value are assigned the value 1 greater than the former value. The initial value has to be constant, or already defined index symbol.

Enumeration constants are a part of C and C++, but with some changes:

- In C, index symbols are accessible from all expressions of same or narrower area of visibility. **Index symbols in AleC++ are accessible if, and only if the enumeration group can be determined from the context.** This allows for the same symbols to be used within two, or more enumeration groups; an ability not found in other languages. An example of this is:

```
Bool status1 = True, status2 = False;
```

but not:

```
int status3 = OK;
```

since one cannot determine from the context which group is involved (`status3` is `int`)interpreter will report an error. This modification was necessary for logic simulation, as enumeration constants are used for logic states. One state can be found in more than one set of possible logic states.

- **Enumeration symbols in AleC++ can be character constants,** since we can determine from the context if the constant is an enumeration one, and which group it belongs to. They do not have ASCII values, but values according to their place in the group. If the interpreter cannot determine from the context if it is an enumeration constant, it will treat it as a character constant.

```
enum digital3 { '0', '1', 'x' };        // index 0, 1, 2
enum digital4 { '0', '1', 'z', 'x'};    // index 0, 1, 2, 3

char c = 'x';           // character - ASCII values
digital3 d3 = 'x';      // enum digital3 - value 2
digital4 d4 = 'x';      // enum digital4 - value 3
```

Enumeration groups of characters are the basis for design of state systems for modelling of digital hardware. Since the symbols are valid only within the group, and bear no influence on other groups, it is possible to form an unlimited number of states, which have the same states (it is realistic to expect states ‘0’ ,’1’, and ’x’ to repeat often.)

- In AleC++, an **enumeration string** can be defined. It does not differ from the common one, and the recognition by the interpreter is done in that usual way, as is the case with the individual constants. Enumeration strings can consist of symbols defined in the appropriate enumeration group.

```
char *s = "string character"; // common string
digital3 *d3s = "0001x1x1";   // enumeration string - digital3
digital4 *d4s = "zzzz1101";   // enumeration string - digital4
digital4 *d4e = "0101xxaa";   // error - 'a' is not in the group
```

Common strings end with the character ‘`\0`’, which is (n+1)st character of a string with length n, although that character is not displayed. Enumeration strings do not have that character at the end since the first symbol in the enumeration group is 0. **To determine length of an enumeration string, one has to use new command** `lengthof` to be explained in the following chapters.

- Longer enumeration strings can be a reading challenge, i.e.:

```
"0001011100xx0011"
```

This string representing a 16-bit word would be readily understandable if bytes, or even nibbles were separated. To that goal AleC++ introduces enumeration separator, a common non-indexed character constant, skipped in enumeration strings:

```
enum digital3 { '0', '1', 'x', 'X' = 'x', ' ' = void, '_' = void };
digital3 es1[] = "0001011100xx0011"; // string without the separator
digital3 es2[] = "0001_0111 00XX_0011"; // string with separators
```

This example shows that multiple separators can be introduced, initialized as **void**. The value of both `lengthof(es1)` and `lengthof(es2)` would be 16, since **the separator does not affect the length of the string**. The first symbol after the separator assumes the index value one larger than the index value of the symbol before the separator **since the declaration of the separator does not affect indexing**.

Note: The example above shows case-insensitivity (both 'x' and 'X' have value 2).

2.2.4. Operators

Operators are symbols that indicate arithmetic, logical, and other operations over symbols-operands. AleC++ supports existing C, and C++ operators, and defines some new ones. These are:

`~&`	binary NAND
`~|`	binary NOR
`~^`	binary XNOR
`<-`	signal assignment
`$`	direct access to formal signals
`$$`	total number of formal signals
`@`	attribute, partial differentials
`ddt`	first time derivative
`d2dt2`	second time derivative
`idt`	time integral
`sdt`	second partial time differential $\left(\frac{d^2 f(x)}{dtdx}\right)$

The new operators were introduced to satisfy the needs for simulation and functional modelling. While the first three are a simple negation of existing ones, the fourth operator makes the basis for modelling of the communication between parallel processes (you can read more in the chapter on digital circuits modelling.) The rest of the operators are going to be discussed in the text that follows.

2.2.5. Separators

The list of the separators will be introduced now while the detailed explanation of their usage will be left for following chapters:

```
[ ]  { }  ( )  : ; , ... #
```

2.3. Preprocessor

Preprocessor is a separate part of the interpreter, which analyses the text, and creates temporary file. Preprocessor commands differ from the others by the symbol “#” situated in the first column. Preprocessor can define macros, with or without the parameters, include other files, or control the parts of the text to be interpreted. Preprocessor of AleC++ supports the standard C preprocessor partially in the following directives:

```
#define
#include
#ifdef
#ifndef
#else
#endif
```

These directives are fully supported, and can be used without limitations. In case of need other directives will be included in the follow-up versions of AleC++.

The `include` command functions with the names of files between characters " and ", or in between characters < and >. Files with names given between characters < and > are searched for in the system directory, defined by the system variable ALEC_HOME (see the installation procedure). Files with names given with " and " are searched for in the current directory.